Functional vs Object-Oriented Distributed Languages*

R.F. Pointon[1], S. Priebe[2], H-W. Loidl[1], R. Loogen[2], and P.W. Trinder[1]

[1] Department of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh, Scotland;
{rpointon,hwloidl,trinder}@cee.hw.ac.uk

[2] Fachbereich Mathematik und Informatik,
Philipps-Universität, Marburg, Germany;
{priebe,loogen}@informatik.uni-marburg.de

Abstract. Conventional distributed programming languages require the programmer to explicitly specify many aspects of distributed co-ordination, including resource location, task placement, communication and synchronisation. Functional languages aim to provide higher-level abstraction, and this paper investigates the effectiveness of this for distributed co-ordination. The investigation contrasts and compares Java and two Haskell-based distributed functional languages, Eden and GdH. Three distributed programs are used as case studies, and the performance and programming effort are reported.

1 Introduction

Improved network technology and a rapidly expanding infrastructure of interconnected computer systems have generated a resurgence of interest in distributed languages. This is demonstrated by the popularity of the modern object-oriented language Java.

Theoretically non-strict functional languages offer additional benefits for constructing distributed systems. These languages have sophisticated, e.g. polymorphic, type systems to further enhance the safety of distributed systems. They also have a highly-dynamic model of distribution with communication on-demand, a relatively high degree of distribution transparency with the runtime-system managing details of communication and synchronisation, and the potential to safely abstract over distribution-control primitives with higher-order polymorphic functions.

These benefits of distributed functional languages are bought at the price of an elaborate implementation. This may explain why, despite many distributed functional language designs over the past decade, robust well supported implementations have only recently become available. These include Erlang [5]

* Supported by UK's EPSRC (grant GR/M 55633), British Council/DAAD (travel grant no. 1097), and the Austrian Academy of Sciences (APART 624).

and Haskell with Ports [10], which we discuss in Section 6. The authors represent groups who have implemented two non-strict distributed functional languages based on Haskell [12], namely: Eden [2] and Glasgow distributed Haskell (GdH) [13].

The paper opens with Section 2 introducing distributed language concepts and describing the distributed programming model underlying Eden, GdH and Java. For the comparison three different applications of increasing size were chosen. The first two applications, *Broadcast Ping* (Section 3) and the *Chat Room* (Section 4) were implemented in all three languages so as to contrast and compare the style of programming. The final *Multi-user Shared Map* (Section 5) application is implemented only in GdH so as to show some features unavailable (or non-trivial to implement) in the other systems. Section 6 relates our work to other distributed functional languages, and finally a summary is presented in Section 7.

2 Distributed Concepts and Languages

2.1 Distributed Concepts

We compare the distributed programming models of Java, Eden and GdH using the following concepts. *Decentralised:* co-operating systems, e.g. client-server, vs *centralised* — a single system, often hierarchical with a single point of failure. *Location Aware:* able to identify and make use of the attributes and resources at each processing element (PE) location, vs *location independent* — where the location is irrelevant or handled automatically. *Fault Tolerant:* able to detect and recover from errors.

Table 1. Language Comparison of Distributed Concepts.

	GdH	Eden	Java
Concepts:			
Paradigm	Functional	Functional	Object Oriented
Decentralised	No	No	Yes
Location Aware	Yes	Partial	Yes
Fault Tolerant	Partial	Partial	Yes
Techniques:			
Resource Lookup	N/A	N/A	Registry
Thread Placement	Remote evaluation	Process	RMI
Synchronisation	MVar & shared data	Implicit	Synchronised class
Communication	MVar & shared data	Channel	RMI
Evaluation	Non-strict	Mixed	Strict
Fault-tolerance	Exceptions	Exceptions	Exceptions

Depending on the concepts supported by the languages they provide certain distribution related techniques. *Resource Lookup:* the means for locating/testing for resources from another location. *Thread Placement:* how work is sent to remote locations. *Synchronisation:* the way of co-ordinating activities. *Communication:* the means for sending and receiving information. *Evaluation:* the execution model of the language, where *strict* results in function arguments being evaluated before entering a function, vs *non-strict* where evaluation is delayed until the arguments are used. *Fault-tolerance:* the method of handling errors. Table 1 summarises and presents the concepts and techniques used by the three languages.

2.2 Languages

Java [6] is well known and only briefly outlined here. It supports distribution via the Remote Method Invocation (RMI) API, which allows Java programs to invoke methods of remote objects. Dynamic class loading allows programs to load code on-demand, in particular, the code can be loaded across the internet. A subset of the commands and classes are shown in Figure 1.

```
\\locks objects of 'expr' to ensure mutual exclusion in 'statement'
synchronised (expr)
   statement

\\to allow remote objects by defining new classes whose :
\\ - interface extends java.rmi.Remote
\\ - implementation extends java.rmi.server.UnicastRemoteObject

\\associate a name with a new remote object
static void java.rmi.Naming.rebind(string name, remote obj)

\\return a reference to a remote object
static remote java.rmi.Naming.lookup(string name)
```

Fig. 1. Java Mechanisms for Distribution.

GdH [13] supports both parallel and distributed computation using two classes of threads: purely-functional threads and explicit side-effecting I/O threads. The parallel concepts are inherited from Glasgow Parallel Haskell (GpH) [14], and the concurrent I/O concepts form Concurrent Haskell.

Remote I/O thread placement with implicit communication of the result is achieved by a new language primitive for remote evaluation (`revalIO`). Communication and synchronisation may be implicit: threads on one PE can share variables with threads on other PEs. Data transfer is lazy: only the immediately required portion of a data structure is transferred when required by another thread. I/O threads can explicitly communicate and synchronise using polymorphic semaphores (`MVars`) and abstractions built on top of them, such as channels

```
-- MVar operations
newEmptyMVar :: IO(MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO()

-- location identification
allPEId      :: IO[PEId]
myPEId       :: IO PEId

-- thread creation and placement
forkIO       :: IO a -> IO ThreadId
revalIO      :: IO a -> PEId -> IO a

-- class of all object types that can be located on a PE
class Immobile a where
      owningPE :: a -> IO PEId
```

Fig. 2. GdH Mechanisms for Distribution.

and buffers. Thread location is exposed to the program level so that a program can use resources unique to a PE.

Furthermore evaluation strategies, as developed for parallel execution in GpH [14], allow control of the evaluation degree and the evaluation order of Haskell expressions. Strategies provide the means for tailoring the non-strict behavior of evaluation as well as communication. We have started to extend evaluation strategies to express distributed co-ordination by specifying both work and data placement. The most important basic strategies and combinators are presented in Figure 3.

```
type Strategy a = a -> ()

-- reduction to weak head normal form
rwhnf   :: Strategy a

-- class of all reducible types with reduction to normal form strategy
class NFData a where
      rnf :: Strategy a

-- strategy application
using   :: a -> Strategy a -> a
usingIO :: IO a -> Strategy a -> IO a

-- work placement
on      :: a -> PEId -> a
```

Fig. 3. GdH Strategies for Distributed Co-ordination.

Eden [2] uses explicit processes together with implicit communication via stream-based channels. Lean communication is achieved by an eager sending protocol and communication connections can be built up flexibly by using dynamic reply channels. Complex process nets can be constructed to realize transformational and reactive systems.

Process specifications are explicitly visible in the language: *Process abstractions* (see first part of Figure 4 for an example) specify which output a process will generate depending on process parameters $\mathtt{p}_i$ (supplied on creation time) and process input $\mathtt{in}_i$ (supplied later via communication). One example is the predefined *merge* process abstraction, which joins a set of streams into a single stream non-deterministically allowing for many-to-one communication. Such an abstraction serves as a template for later *process instantiation*: When a process abstraction is applied to a full set of input parameters via the `#` operator, a process will be created. The system function `noPE` can be used to query the runtime system for the number of available PEs. This is useful as it is advisable to relate the number of processes to the number of available PEs.

As future work, a system function similar to the GdH `myPEId` and an extension of the process instantiation operator (`#`) for direct process placement could easily be added to Eden to support more location-awareness.

$-- \, process \ abstraction$
$\mathtt{pabs} \ :: \ pt_1 \rightarrow ... \rightarrow pt_k \rightarrow \textbf{Process} \ (it_1, ..., it_m)(ot\_1, ..., ot_n)$
$\mathtt{pabs} \ \mathtt{p}_1 \ \ldots \ \mathtt{p}_k = \textbf{process} \ (\mathtt{in}_1, \ldots, \mathtt{in}_m) \rightarrow (\mathtt{o}_1, \ldots, \mathtt{o}_n)$
$\qquad\qquad \textbf{where} \ \ldots$

$\mathtt{merge} \ :: \ \textbf{Process} \ [[a]] \ [a]$

$-- \, process \ instantiation$
$\mathtt{(\#)} \ :: \ (Transmissible \ a, Transmissible \ b) \Rightarrow Process \ a \ b \rightarrow a \rightarrow b$

$\mathtt{noPE} \ :: \ Int$

Fig. 4. Eden Mechanisms for Distribution.

3 Broadcast Ping

Broadcast Ping is similar in nature to the UNIX ping command in that it determines the round trip time between machines. This particular version of ping determines the time necessary to instantiate a remote thread, perform a trivial environment variable lookup to confirm the location, and return this result. A distinguished, or main, PE performs this process for every PE and returns a list of locations and timings. The ping program demonstrates remote thread creation; access to a remote resource; and communication from remote threads.

3.1 Programming Style

Table 2 summarises the program size for each language. We separate the computational code from the code for the co-ordination of the distribution and comment on language specific features of the implementations.

The GdH and Eden programs share the same structure with only the methods for determining the PEs and creating a remote task changing. Java uses a decentralised model and therefore the program is split into two components (client and server) together with a common interface. The greater size also results from the fact that Java forces the programmer to consider exceptions when defining methods, i.e. to implement fault-tolerance — though this may be a design advantage when compared to other languages where the programmer often delays implementation of exception handling until it is required.

Table 2. Ping Programming Style.

	GdH	Eden	Java
Lines of code	16	17	37
Lines for distribution	2	3	4
Other comments	—	—	Fault-tolerant

3.2 Performance

The program in each language is executed on a network of four machines using `ushas` as the main PE. Table 3 gives the execution times. Where a remote PE is accessed, communication costs dominate and there is no significant difference between the languages. However, the two centralised languages GdH and Eden are faster in the case where communication is local, because the Java program has separate client and server programs and must perform inter-process communication and registry lookup to connect.

Table 3. Ping Performance (times in milli-seconds).

	GdH	Eden	Java
ushas (local)	0	0	2
bartok	3	4	3
ncc1705	2	3	2
brahms	3	3	2

3.3 Discussion

The differences between centralised and decentralised languages are very apparent from this program in the structure of the program, though the actual performance is similar for all systems. Java possesses a potential advantage in that the enforced exception handling results in a more robust program, whereas the other systems crash if a PE fails during a ping.

4 Chat Room

The Chat Room is a client server type application, where multiple users chat to each other through a message board maintained by the server. The program also has additional extensions for maintaining simulation information at the server and a nominated super user is allowed to manipulate it. The Kiteck simulation is discussed in [11] and the environment is implemented in Java as described in [16]. The client server nature demonstrates the instantiation of remote threads; the use of shared immobile resources for communication; the creation of individual GUIs for users to interact with the client threads; and fault-tolerance in handling of remote GUI errors.

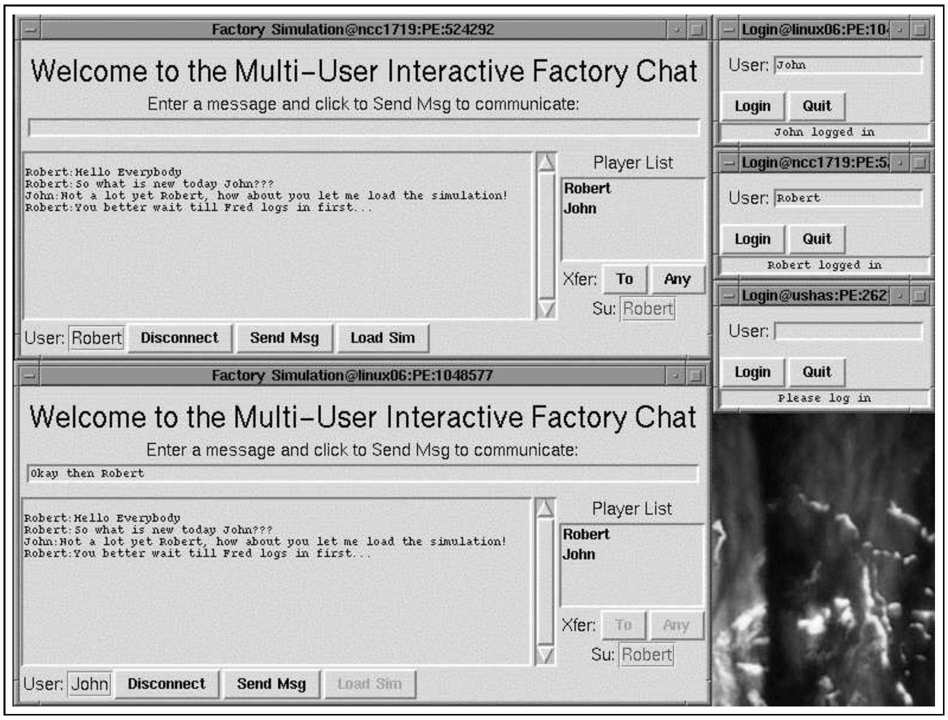

Fig. 5. The Chat Room Interface. (With all X screens re-directed to one host.)

4.1 Programming Style

We have investigated two alternative implementations of the system, one with the server represented as *shared synchronised state*; and one with a *server as a process* managing its internal state and operating on a queue of messages from the clients. Table 4 summarises the program size and details in each language.

Table 4. Chat Room Programs.

	GdH	Eden	Java
Lines of code	500	≈ 450	1500
Lines for distribution	15	5	15
Other comments	—	(Incomplete)	Fault-tolerant

Shared Synchronised State is where the server stores the state and the runtime system maintains the state ensuring consistency, mutual exclusion of access and lack of deadlock, etc. The existing Java implementation used a server as shared state and the GdH implementation duplicated this model. Java allows the programmer to mark classes as "synchronised" to enforce mutual exclusion of access to the objects of that class, after which no additional effort is needed by the programmer. GdH required the user to setup an `MVar` to hold the state and then use the code from Figure 6 (similar to what exists in the Haskell semaphore libraries) to implement mutual exclusion and create a function `lock` through which all access can take place.

```
lock :: MVar a -> (a -> IO(a,b)) -> IO b
lock sync update = do
  v <- takeMVar sync
  (nv,res) <- update v   -- apply the update operation
  putMVar sync nv
  return res
```

Fig. 6. GdH Implementation of a Mutual Exclusion Lock.

Server as a Process requires the programmer to implement a process which explicitly manage its external state in response to messages received. Thus the programmer is responsible for defining the access protocol unlike shared synchronised state where the protocol is defined in the runtime system.

In Eden the server and each client are implemented as a process which changes state according to the protocol when receiving messages. Communication is implicit in Eden, and message queues are modeled as lazy lists, thus the communication operations are therefore replaced by list operations. The central part of the Client process shown in Figure 7 is the function `cloop`. It handles incoming messages from the Server, changes the Client's state, and sends new messages to the Server according to the communication protocol.

```
client = process toClients → toServers
  where (_, toServers) = unsafePerformIO (cloop cInitState toClients)

    cInitState = ([], initialSU, CNone) -- intial client state

    cloop :: (ClientState,[ToClient]) → IO (ClientState,[ToServer])
    cloop (s@(out, su, state), tcs) = do
      ...-- process messages

server = process toServers → toClients
  where...-- similar structure to client
```

Fig. 7. Eden Client and Server Structure.

The `unsafePerformIO` in `client` is introduced to allow side-effecting I/O operations like GUI control, whereas in GdH every remote computation is enclosed in an I/O thread by default.

```
main = print fromClientsM -- demand to generate processes
  where
    fromClientsMerge = merge fromClients
    fromClients = [client # tc | tc ← toClients]
    toClients = map (CEmpty:) (server fromClientsMerge)
```

Fig. 8. Eden Process System Instantiation.

The process system itself is constructed via a set of mutually dependent list definitions in Figure 8. The predefined merge process is used to join the client streams into a single input stream for the server. The ease of creating process nets comes at a cost: Eden currently lacks the ability to specify a particular PE with the runtime system using a round robin scheme to choose a process PE.

4.2 Discussion

The GdH and Java implementations are similar though the Java is far more verbose. The Eden implementation reveals the ease at which the process creation can be abstracted away and defined at a higher-level, thus vastly simplifying all the communication and process creation issues. The lack of location awareness is highlighted in Eden as there is currently no way to specify which PE to place the process on, this could be a potential handicap for some programs where the programmer wants something to occur on a particular PE.

5 Multi-User Shared Map

The Multi-User Shared Map is another client server type application with multiple users interacting under the co-ordination of the server. Maps of geographical information systems (GIS) in ARC [3] format are used to provide the data — in this case the maps are comprised of approximately 150,000 drawing lines. The program demonstrates how a large shared data structure can be used implicitly from multiple PEs in the following ways. Unique resources on different PEs are used to generate a single monolithic data structure. Scalability gained by spreading the structure across multiple PEs. Parallel co-ordinated data access as multiple users interact simultaneously with the data.

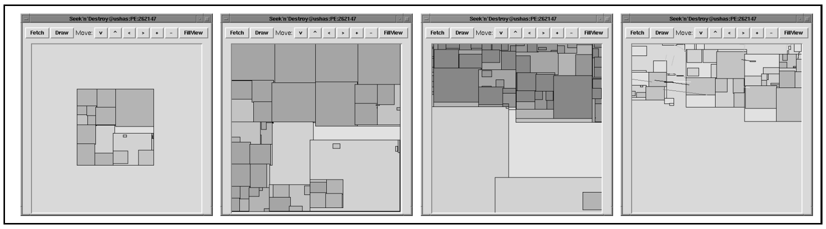

Fig. 9. Zooming into the Map.

5.1 Programming Style

This program is written only in GdH to illustrate some of the communication control strategies supported by the language. The core GdH program is shown in Figure 10 and proceeds as follows. The `main` function (on the main PE) reads in the quadtree map. A GUI running `window` is mapped across all the available PEs whilst sharing the quadtree structure. The sub-function `showit` then is invoked by the GUI to show a particular region which it does by determining the visible components of the quadtree via `visible` and then drawing it to the GUI with `draw`.

The function `window` implements each remote GUI and in doing so specifies the way how the communication of the quadtree from the main PE should be performed. We have investigated the following alternatives for the communication. In each case the only modification in the code affects the strategy applied to the (shared) data structure that is to be transferred.

Implicit Sharing: the data is transferred on the demand from each of the remote windows. This requires no change to the original code.

```
main = do
  q ← readQuad "map" ‘usingIO‘ rnf [a]
  pes ← allPEId
  mapM (rforkIO (runGui (window q))) pes
window q gui = do
  init gui
  ...
  showit b = do
    let m = visible q b
    draw gui m ‘demanding‘ rnf m [b]
```

[a] The stragegy `‘usingIO‘` is used to ensure that the entire quadtree is read in and processed before the program continue.

[b] The `‘demanding‘` stategy on the draw is used to ensure that the visible components of the quadtree have been fully evaluated before sending them to the GUI, this avoids the problem of the GUI blocking while it does the evaluation.

Fig. 10. Structure of Basic Map Program.

Eager Transfer: transfer a copy of the entire quadtree to the remote PE before the GUI continues. The changes required for this are shown in Figure 11.

Explicit Functional Communication: send work requests to the main PE where the quadtree is and let the main PE calculate the visible components and return this result to the GUI. The changes to control the visible evaluation are shown in Figure 12.

Explicit I/O Communication: explicitly communicate with a server process on the main PE, sending a request for the server to calculate the visible result, then wait for the result. In this case the remote GUI is started with a shared reference to a communication channel (`ch`) to the server rather than a shared quadtree. The introduction of the explicit I/O can be seen in Figure 13. This is identical in approach to the explicit functional method yet implements it using I/O commands in the style more similar to traditional languages.

5.2 Performance

In measuring the performance of the different strategies for communication, timing code is placed around the communication. The results collected are at each step as the user zooms into the map, as in Figure 9, and then reverses and zooms out, thus traversing the same data space of the map. In Figure 14 these discrete steps of traversal are plotted along the x-axis, and the time in milli-seconds for each step is plotted along the y-axis.

The two explicit approaches yielded identical results which was unsurprising since they are using the same technique, therefore they are only shown once on the graph. From Figure 14 it can be seen that the eager approach has a huge startup cost for the first step but after that it has the lowest map traversal times.

```
window q gui = do
  init gui `demanding` rnf q ^a
  ...
  showit b = do
    let m = visible q b
    draw gui m `demanding` rnf m
```

[a] Demand the full quadtree when the window is initilised.

Fig. 11. Eager Approach for Map Sharing.

```
window q gui= do
  init gui
  ...
  showit b = do
    let m = visible q b
    draw gui m `demanding` (rnf ((m `using` rnf) `on` mainPE)) ^a
```

[a] Evaluate the `visible` function `` `on` `` the main PE and then demand the result back at the GUI.

Fig. 12. Explicit Functional Map Communication.

```
main = do
  ch ← newChannel
  q ← readQuad "map" `usingIO` rnf
  pes ← allPEId
  mapM (rforkIO (runGui (window ch))) pes
  let server = do
    respond ch (\ b → return (visible q b) `usingIO` rnf) ^a
    server
  forkIO server

window ch gui = do
  init gui
  ...
  showit b = do
    m ← request ch b ^b
    draw gui m `demanding` rnf m
```

[a] The forked server thread calculate the visible components in response to the requests it receives.
[b] The GUI sends off a request down the channel to the server.

Fig. 13. Explicit I/O Map Communication.

The explicit approaches are much slower due to always having the communication overhead. The sharing approach varies considerably but the sharing causes it to eventually match the fast eager times.

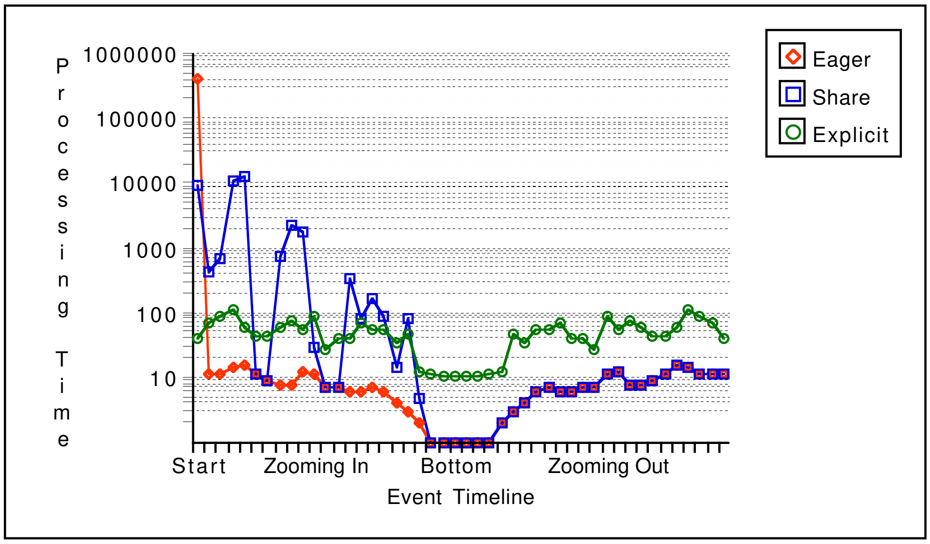

Fig. 14. The Effect of Different Communication Strategies.

5.3 Discussion

The implicit sharing technique required no changes to the code and is often sufficiently fast. We would expect that the sum of the times to traverse the entire data space using the sharing method would be equal to the sum of the times via the eager approach — the communication may be more spread out, but no more communication is needed. For the explicit approach the sum of the times would be higher as the communication must always be repeated. It proves straightforward to code alternative communication schemes using different strategies in GdH.

6 Related Work

Modern distributed languages often make use of advanced concepts for the computation component of the language by using class hierarchies with inheritance, strong typing or automatic garbage collection. However, their control component usually relies on explicit communication and task placement with few facilities for abstracting over these basic constructs. In this section we relate our languages, Eden and GdH, to those distributed functional languages that provide some abstraction facilities over their basic co-ordination constructs. A more detailed survey of distributed extensions of Haskell is given in [15].

Haskell with Ports [10] is a library for Concurrent Haskell adding additional monadic commands for communication between PEs. This results in a decentralised distributed language with explicit communication and moderate support for fault-tolerance. Ports are an abstraction over UNIX-style sockets, realising dynamically typed uni-directional communication channels. Ports are first-order values in the language and the powerful Haskell monad libraries can be used to build abstractions over ports.

ERLANG [1] is a strict, impure, first-order, untyped language with explicit communication constructs. In our classification ERLANG qualifies as decentralised, location-aware with explicit port-based communication. ERLANG has been developed and is used in telecommunications applications and therefore emphasises important pragmatic issues such as real-time fault tolerance via time-outs and exceptions, hot loading of new code, message authentication etc.

In a distributed language logic variables can be used as shared single-assignment variables in order to realise bi-directional communication channels. Several languages such as Distributed Haskell [4] and Curry [7] use this construct to obtain a declarative co-ordination component of their language. Similar constructs are used in OZ [8], a distributed language combining functional, object-oriented, and logic paradigms for computation as well as co-ordination. Finally, Brisk [9] is based on a virtual shared heap, similar to GdH, in order to support implicit communication.

7 Summary

We have compared two distributed functional languages, Eden and GdH, with Java using three programs. Java's distribution model is decentralised, and hence is the most powerful, followed by GdH which is location aware, followed by Eden which is neither. Although both the Eden and GdH versions of the programs are significantly shorter than the Java version, only in Eden is the distributed co-ordination higher-level than Java. In effect the distributed co-ordination available in Java and GdH is at a similar level of abstraction. However, it is easiest to abstract and compose co-ordination constructs in GdH, e.g. to construct a lock from an `MVar`. The value of separating of co-ordination and computation concerns has been established for parallel programming [14], and we find that it also aids distributed programming. For example, it is easy to specify alternative communication strategies in the multi-user shared map. Performance of all three languages is similar for a simple program (ping), although tuning is easiest in GdH, merely requiring changes to the strategies.

In terms of selecting a language for a given application, the following tentative guidelines are proposed. If a dynamic client server model with fault-tolerance is required then use Java. If large complex data structures are to be shared then use GdH. If the problem has a regular process model and is mostly purely functional then use Eden.

References

1. J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikstrom. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Programming Models*, pages 120–124. IEEE Computer Science Press, Geneva, Switzerland, April 1997. Available from [17].
3. P. Burrough and R. McDonnell. *Principles of Geographical Information Systems*. Oxford University Press, 1998.
4. M.M.T. Chakravarty, Yike Guo, and M. Köhler. Distributed Haskell: Goffin on the Internet. In *Fuji Intl. Symposium on Functional and Logic Programming*, pages 80–97. World Scientific Publishers, 1998.
5. Claes Wikstrom. Distributed Programming in Erlang. In *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Linz, September 1994.
6. M.C. Daconta, A. Saganich, E. Monk, and M. Snyder. *Java 1.2 and JavaScript for C and C++ Programmers*. John Wiley & Sons, New York, 1998.
7. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *PPDP'99 — Principles and Practice of Declarative Programming*, Paris, France, September 1999.
8. S. Haridi, P. van Roy, and G. Smolka. An Overview of the Design of Distributed Oz. In *PASCO'97 — Intl. Symposium on Parallel Symbolic Computation*, pages 176–187, Kihei, Hawaii, July 1997. ACM Press.
9. I. Holyer, N. Davies, and E. Spiliopoulou. Distribution in a Demand Driven Style. In *COCL'98 — Intl. Workshop on Component-based software development in Computational Logic*, pages 29–41, University of Pisa, Italy, September 1998.
10. F. Huch and U. Norbisrath. Distributed Programming in Haskell with Ports. In *IFL'00 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 2011, pages 107–121, Aachen, Germany, September 2000. Springer-Verlag.
11. P. Kiteck. Analysis of Component Interaction in a Distribution Facility using Simulation. In *EUROSIM*, 1992.
12. S.L. Peyton Jones, J. Hughes, et al. Haskell 98: A Non-strict, Purely Functional Language. Available at <URL:`http://www.haskell.org/`>, February 1999.
13. R.F. Pointon, P.W. Trinder, and H-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *IFL'00 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 2011, pages 53–70. Springer-Verlag, September 2000. Available from [18].
14. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.
15. P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells. *J. of Functional Programming*, 2001. To appear.
16. P.A. Wilcox. Object-Oriented Factory Simulation in Java. Master's thesis, Department of Computer Science, Edinburgh University, 1997.
17. Eden Web Pages. <URL:`http://www.mathematik.uni-marburg.de/inf/eden`>.
18. GdH Web Pages. <URL:`http://www.cee.hw.ac.uk/~dsg/gdh`>.